HW3

黄佳溢

November 19, 2024

1 实验配置以及代码理解

我本次使用的是外接显示屏，并在内核配置中打开了Frame Buffer设置。同时为了深入理解代码的实现逻辑，这里我就结合课程内容，来完整描述显示屏接入后的流程:

1. **显示屏刚接入**
   显示屏作为**字符型设备**，接入之后内核首先会调用显示屏驱动中的init函数申请设备号并保存，此时便完成了设备的初始注册。

2. **建立inode**
   我们的开发板内核已经默认打开了udev并支持Frame Buffer，此时便会自动在/dev目录下建立fb0字符设备文件。

3. **访问并绘制**
   我们在代码中只需要调用open()函数便可以获取相应的设备描述符fd，并调用(unsigned char *)mmap向内核内存空间申请一块空间作为映射空间，至此，我们就可以对这块内存做读写操作来代替直接的IO操作！

2 软件结构

考虑到所有的画图程序的基本流程为：

$$\backslash dev\backslash fb0 \xrightarrow[\text{mmap}]{\text{open}} \text{fd,fbp} \xrightarrow{\text{ioctl}} \text{vinfo,finfo} \rightarrow \text{offset} \rightarrow \text{Draw}$$

为了简化程序结构，我将$\backslash dev\backslash fb0 \xrightarrow[\text{mmap}]{\text{open}} \text{fd,fbp}$这一部分在**主函数**中实现，并将fd和fbp**作为传参输入到基本函数中**，后面的部分交由基本函数来执行。整个软件结构如图：

```
├─── app
│    └─── move_circle.c
├─── image
│    └─── bear.bmp
├─── include
│    ├─── App.h, Pic.h
├─── main
│    ├─── bmp_show.c, main.c, move.c
├─── Makefile
└─── tools
     ├─── circle.c, curve.c, dot.c, ellipse.c, line.c
```

我们将主函数均放在**main**目录下，将基本函数放在**tools**目录下，将库函数放在**include**目录下，将应用程序放在**app**目录下，将图片放在**image**目录下。

3 主函数

这个是最简单的操作，在课件上也讲解地足够详尽，这里就简单展示代码：

```c
int main(void)
{
    struct fb_fix_screeninfo finfo;
    struct fb_var_screeninfo vinfo;
    unsigned char *fbp = 0;
    int fd;
    int offset;
    int buffersize;

    fd = open("/dev/fb0", O_RDWR);
    if(fd==-1){
    printf("fd is error!\n");
    return 1;
    }

    ioctl(fd, FBIOGET_VSCREENINFO, &vinfo);
    ioctl(fd, FBIOGET_FSCREENINFO, &finfo);
    printf("Now you have got vinfo and finfo\n");

    printf("=====================================\n");
    printf("Line length is %d\n", finfo.line_length);
    printf("Yres is %d\n", vinfo.yres);
    printf("Xres is %d\n", vinfo.xres);
    printf("=====================================\n");
```

Figure 1: 主函数

```c
buffersize = finfo.line_length*vinfo.yres;
printf("Buffersize is %d\n", buffersize);

if((fbp = (unsigned char *)mmap(
    NULL,
    buffersize,
    PROT_READ|PROT_WRITE,
    MAP_SHARED,
    fd,
    0
))==-1){
printf("fbp is error!\n");
return 1;
}

memset(fbp, 0, buffersize);
printf("Begin to draw!\n");
```

Figure 2: 主函数

在代码中，我使用printf函数打印了设备的基本信息，便于在测试和使用时查看设备的状态。

4 基本函数

当我们进行绘图的时候，必须提前得知显示屏的基本参数（如RGBA占用的bit长，长宽），这些信息都可以通过输入**fbset**来获取。得到的输出为：

```
mode "1024×600-0"
rgba 8/16, 8/8, 8/0, 0/
```

从中可以得到以下信息：①显示屏长1024，宽600 ②24位色深，RGB分别占1个字节 ③从低位地址到高位地址分别经过BGR

4.1 画点、线和任意曲线

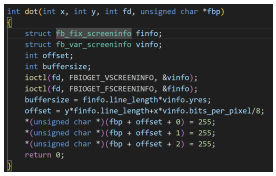

Figure 3: Dot.c

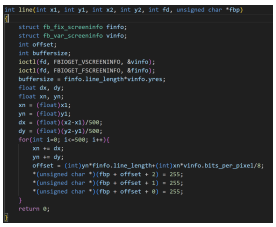

Figure 4: Line.c

Figure 5: Curve.c

代码逻辑上还是比较简单易懂的，我简单提及几个要点：

1. **传参**

   在所有程序中，传参均包含了fd和fbp，基本逻辑是： fd$\longrightarrow$vinfo, finfo$\longrightarrow$line_length, bits_per_pixel$\xrightarrow{x,y}$offset$\xrightarrow{fbp}$Operand

$$\text{offset} = y * \text{finfo.line\_length} + x * \text{vinfo.bits\_per\_pixel}/8; \tag{1}$$

$$\text{Operand} = \text{fbp} + \text{offset} \tag{2}$$

2. **画点**

   通过上面的流程，可以得到操作数所在的地址Operand，对指向地址进行赋值。

$$*(unsigned\ char*)(\text{Operand} + 0) = \textbf{Blue} \tag{3}$$

$$*(unsigned\ char*)(\text{Operand} + 1) = \textbf{Green} \tag{4}$$

$$*(unsigned\ char*)(\text{Operand} + 2) = \textbf{Red} \tag{5}$$

3. **画线**

Algorithm 1: Draw line

input : $(x_1, y_1), (x_2, y_2), fd, fbp$

output: Draw a line on screen

Obtain Operand;

$dx \leftarrow (x_1 - x_2)/500$, $dy \leftarrow (y_1 - y_2)/500$;

$x \leftarrow x_2$, $y \leftarrow y_2$;

while $x \leq x_1$ **do**

    Dot$(x, y, color)$;

    $x = x + dx$, $y = y + dy$;

end

4. **画曲线**

   和画线其实区别不大，都是通过dx, dy来实现逐个画点，唯一的区别就在于，画曲线会要求输入函数$(*op)()$，这样一来，每个循环中，我们绘制的位置就是$(x, op(x))$，然后调用Dot$(x, op(x), color)$即可完成绘制。这里就不放伪代码了。

下面展示一些绘制的结果:

Figure 6: 画线

Figure 7: 画曲线

4.2 画圆，椭圆并移动

我们首先实现绘制定点圆及椭圆，需要调用math.h库函数，伪代码如下：

Algorithm 2: circle.c

input : $(x_0, y_0), r, a, b, fd, fbp, color$

output: Draw a circle or ellipse on screen

$ang \leftarrow 0$;
$\Delta ang \leftarrow \frac{2\pi}{100}$;
while $ang \leq 2\pi$ **do**
    $x \leftarrow x_0 + a \cdot r \times \cos(ang)$;
    $y \leftarrow y_0 + b \cdot r \times \sin(ang)$;
    $\mathrm{Dot}(x, y, color)$;
    $ang \leftarrow ang + \Delta ang$;
end

当输入满足$a = b$时，绘制出来的图形就是圆，当$a \neq b$时，绘制出来的椭圆中，a是长轴，b是短轴。

接下来我们就利用该基本函数来构建我们的move_circle.c函数，不失一般性，我构建两个圆，圆心位置相对中心点对称：

Algorithm 3: move_circle.c

input : $(begin\_x, begin\_y), (end\_x, end\_y), r, fd, fbp$

output: Draw two moving circle

$dx \leftarrow (x_1 - x_2)/500$, $dy \leftarrow (y_1 - y_2)/500$;
$x \leftarrow x_2$, $y \leftarrow y_2$;
while $x \leq x_1$ **do**
    $circle(x, y, r, fd, fbp, white)$;
    $circle(1024 - x, 600 - y, r, fd, fbp, white)$;
    Delay for sometime.;
    $circle(x, y, r, fd, fbp, black)$;
    $circle(1024 - x, 600 - y, r, fd, fbp, black)$;
    $x = x + dx$, $y = y + dy$;
end

可以看到，move_circle.c程序中采用**“将圆形覆黑”**的操作来完成前一时刻图形的擦除，相比较$memset(fbp, 0, buffersize)$，速度更快。

5 BMP图形文件显示

BMP图形文件采用无压缩的图片格式，我们可以通过file命令来获取文件的基本信息：

```
(base) xm@MI: $ file bear.bmp
bear.bmp: PC bitmap, Windows 3.x format, 253 x 155 x 24, cbSize
117854, bits offset 54
```

可以看到，我们的图片是253 × 155，色深24位，从低位到高位分别是BGR，像素偏移54bits（这代表我们从54bits开始读取像素值，前面的是图像的基本信息）。同时我们还需要注意BMP的一个存储特性：**每一行的像素字节数必须得是4的倍数，少了就补齐**。这在后面的程序设计中非常重要！该程序设计，重点在于BMP文件的读取和利用，最后调用Dot函数就可以绘制图像。下面我仅介绍如何读取BMP文件：

$$\text{bear.bmp} \xrightarrow{\text{fopen}} \text{file} \begin{cases} \xrightarrow{\text{fseek(10)}} \text{pixel\_offset} \\ \xrightarrow{\text{fseek(18)}} \text{width} \\ \xrightarrow{\text{fseek(22)}} \text{height} \end{cases}$$

为了避免内存溢出，我只申请了**行字节长度**的内存空间，每次只从file中读取一行的数据并进行绘制。行字节长度定义如下：

$$\text{row\_size} = (\text{width} * 3 + 3)\&(\sim 3)$$

这样可以保证row_size刚好是4的倍数。之后的内容就是老生常谈了，每次读取一行数据，使用Dot绘制。

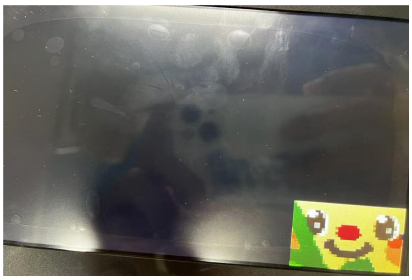

Figure 8: 效果

Figure 9: 原图

6 Makefile，静态动态库

我们在程序中使用Makefile对程序进行管理，Makefile的基本流程为：先将基本函数全部打包合成静态库libdraw.a，接着我们链接主函数时直接带上即可。

```
CC=loongarch64-linux-gnu-gcc
TARGET_LIB=libdraw.a

SRC_DIR=./tools
MAIN_DIR=./main
HEADER_DIR=./include

SRCS=$(SRC_DIR)/dot.c $(SRC_DIR)/line.c $(SRC_DIR)/circle.c $(SRC_DIR)/ellipse.c $(SRC_DIR)/
OBJS=$(SRCS:.c=.o)

$(TARGET_LIB): $(OBJS)
    @echo "Creating static library $(TARGET_LIB)..."
    ar rcs $@ $(OBJS)

$(SRC_DIR)/%.o: $(SRC_DIR)/%.c
    $(CC) -c $< -o $@ -I $(HEADER_DIR)
```

Figure 10: Makefile

```
clean:
    @echo "Cleaning up....."
    rm $(OBJS) $(TARGET_LIB)

Main: $(TARGET_LIB) $(MAIN_DIR)/main.c
    $(CC) $(MAIN_DIR)/main.c -o Main -L. -lm -ldraw -I $(HEADER_DIR)

Move: $(TARGET_LIB) $(MAIN_DIR)/move.c
    $(CC) $(MAIN_DIR)/move.c app/move_circle.c -o Move -L. -lm -ldraw -I $(HEADER_DIR)

Bmp_show: $(TARGET_LIB) $(MAIN_DIR)/bmp_show.c
    $(CC) $(MAIN_DIR)/bmp_show.c -o Bmp_show -L. -lm -ldraw -I $(HEADER_DIR)
```

Figure 11: Makefile

6.1 静态库，动态库和软件

我们编写一个.c文件并希望将其转化为可执行文件，需要经历以下步骤：预编译（将#include展开），编译（检查语法），汇编（转化为机器语言），链接（打包代码中用到的库文件）。静态库和动态库的产生和使用就在最后一步**链接**。

静态库链接是将整个库文件和目标文件一起打包构成了可执行文件，占用空间大，并且当其中的某个模块修改后，就得重新打包和编译整个文件，但是使用方便，最后仅需要一个可执行文件。在我们的Makefile中，具体的指令如下（仅包含dot.c）：

```
(base) xm@MI: $ loongarch64-linux-gnu-gcc -c dot.c -o dot.o -I ./in-
clude
(base) xm@MI: $ ar rcs libdraw.a dot.o
(base) xm@MI: $ loongarch64-linux-gnu-gcc ./main/main.c -o Main
-L. -ldraw -lm -I ./include
```

动态库是可执行文件在运行中加载执行的，也就是说 程序运行环境中要有动态库文件。一般动态库文件命名为lib***.so。动态库的优点就是方便升级，动态库变化了，可执行文件不用重新编译。在本次实验中并未使用，故并没展示。

7 总结

由于本次实验所设计的程序众多，为了实验报告的整体美观，所以并没有全部展示，我将整个工程文件压缩包一并作为附件上传，里面还包含了两移动圆的视频，感兴趣可以自行观看。

8 参考

静态库动态库
BMP文件格式详解